Compaction of Large Class Hierarchies in Databases for Chemical Engineering

M. Baumeister[1,2] and M. Jarke[2]

[1] Lehrstuhl für Prozeßtechnik
RWTH–Aachen
Turmstraße 46,
52056 Aachen, Germany

[2] Lehrstuhl für Informatik V
RWTH–Aachen
Ahornstraße 55,
52056 Aachen, Germany

{baumeister,jarke}@informatik.rwth-aachen.de

Abstract. Engineering knowledge is characterized by variant and alternative designs for each problem. To alleviate reuse these variants should be captured as classes in the schema of the engineering database. The resulting class hierarchy may be bloated by intersection classes. This paper proposes a mechanism, called "aspects", which uses a restricted form of multiple instantiation to avoid intersection classes. Unlike similar approaches, such as "roles", our work allows various interdependencies between the different aspects of a class to be modelled. We demonstrate these abilities on an example taken from chemical process engineering.

1 Introduction

The use of object–oriented databases and object–oriented modelling has become common in engineering and scientific applications because of the ability of object–oriented models to capture facets of domain knowledge in a concise and intuitively understandable way. However, there are areas which can not yet sufficiently be modelled with standard object models [AWSL92, SLL92, NR92].

One of these areas is the growing variety of variants existing for many engineering artifacts. This variety can be observed within the plastics industry as multi-criteria selection tables, as one-of-a-kind products in mechanical and chemical engineering, or as the thick sales catalogues for new cars. While some of this variety is best resolved within the instance layer, one might be inclined to capture other parts of it within the class layer, in order to partition the database, show connections, support a hierarchical search, and – most importantly to facilitate correct reuse. Expressing variety at the class level of standard class–based object models implies a high number of classes since each variant has to be represented as an own class.

This paper focuses on representing variants occurring in object-oriented simulation models for chemical process engineering. Such models are becoming increasingly important, as the chemical industries try to meet simultaneously the pressure of global competition and stricter environmental pollution standards. While traditional commercially available models stop at the level of relatively large processing units (e.g. reactors, distillation columns), it becomes now necessary to model a chemical process down the level of individual chemical phases[1]. These phase models are translated into mathematical equations which again are numerically solved to answer various interesting questions about the modelled process[2]. An overview about the various models involved and their possible representations is given by [Mar96].

Since directly deriving a large set of equations (up to 100,000) is far from easy, the Chemical Engineering group at RWTH Aachen has developed the VeDa data model which has been formalized and implemented in cooperation with our group, in connection with a Collaborative Research Centre (SFB) called IMPROVE, which has recently been started at RWTH Aachen. VeDa features conceptual models up to the phase–level which are then translated into mathematical equation systems [Mar96]. It is based on VDDL [BM98] a frame–based, object–oriented data definition language. Conceptual model building blocks as well as equation skeletons are represented as classes, since the approach followed by VeDa aims at building a class hierarchy which captures a growing amount of domain knowledge, in order to facilitate reuse of modeling knowledge. In contrast to other languages and database tools for chemical engineering, classes are used up to a very detailed level of attribute refinement such that we usually encounter rather few objects per class. First implementations of VeDa subsets (ModKit [BLM96] and TechMod [DLP+96]) used 'standard' object–oriented

[1] Phases are mixtures of substances sharing the same physical state–combination, pressure, temperature, etc. . E.g. when cooking pasta you have two phases in the pot: a liquid one consisting of the boiling water and the solved salt, and a solid one containing the various ingredients the pasta consists of.

[2] E.g. how many minutes of boiling does it take to solve all of the pasta material in the water.

models, mapped down to relational database technology to reach industrial strength for real-world experiments. Our experiences with these applications showed that either the class hierarchy has to be down–sized considerably, or the amount of classes becomes unmanageable, or we have to model chemical engineering knowledge at so shallow a level that significant knowledge reuse is not happening and modelers will repeat the same modeling mistakes over and over.

In this paper, we therefore propose a new mechanism, called aspects, that is intended to capture richer knowledge while avoiding class explosion. It is aimed to be implementable on top of commercial object–oriented databases and constraints itself to structural object models, since models in our application domain feature (nearly) no methods. In contrast to competing proposals, aspects also addresses the topic of inter-aspect constraints in a principled manner. In Section 2 we illustrate the application and the need for aspects by two relatively simple examples, one from the car domain which may be more familiar to the reader, the other directly from a chemical engineering case study. In Section 3 we define aspects more formally as a new abstraction mechanism for OODBs, and discuss their application constraints. We also report briefly on our implementation of this extension, and on experiences in re-defining the VeDa language using aspects. Section 4 compares this technique to related work and Section 5 concludes this paper.

2 Motivating Examples — How Does Variant Variety Manifest Itself?

We give a short introduction using an imaginary car database, before the full-fledged example from chemical process engineering is presented.

An imaginary car database is intended to hold descriptions of car models. Instead of defining a class "car model" with several attributes and instantiating all car models from it some of the variability shall be captured at the class–level. Therefore three variablity areas are chosen: the type of the model, the manufacturer and the intended usage. In each area several choices exists: Saloon, Estate, Coupe, Sportscar or Roadster are types; VW, Opel or Mercedes are manufacturers; prestige object, family car or small transport are possible usages. Each of these choices may introduce additional attributes or restrict existing ones[3]. For example all Volvos contain two side airbags, the body of Audis is of type "galvanised", roadsters must provide an attribute for the thickness of the tires and family cars always contain back seats. Each car model is then an instance of a combination of the three areas, for example all "Mercedes S-Klasse" variants (still distinguished e.g. by different motor strengths or interiors) are instances of "Saloon–Mercedes–PrestigeObject" whereas most "VW Passats" variants are instances of "Estate–VW–FamilyCar".

Furthermore there might exist dependencies between choices in different variants (e.g. Roadsters are always prestige objects) and the same attribute might be introduced by more than one choice (e.g. Sportscar and PrestigeObject introduce the attribute "steering stabilisation". It has the type "Spoilers" for Sportscars and "ESP" for PrestigeObject). The aim is to represent such a class variety without using an own class for each combination of choices since that would inflate the class hierarchy.

2.1 Modelling a Distillation Column

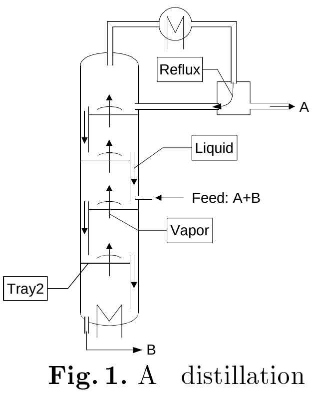

Fig. 1. A distillation column

We now show in detail how variability exposes itself in chemical process engineering. We use a small part of a distillation column model as an example (Figure 1). Distillation columns are used to split a mixture of substances (A+B), most often the result of a reaction, into two or more streams. The mixture enters the column at the feed location in liquid form and is split in the column. The low–boiling components (A) leave the column at the top and the high–boiling ones (B) at the bottom.

To get precise predictions about the purity of the substances leaving at the top or bottom detailed mathematical models are needed, usually in the form of (linearized) differential equations.

Figure 2 shows some classes necessary to represent the behaviour of each single tray in the column. The variability areas are shown as hexagons (we will later call them "aspects"), the choices in each area as rectangles (classes). The classes contain information which can be used by an equation generator to semiautomatically generate the equation system that simulates the behaviour of the specified column (cf. [BM97]). The "aspects" of the class `DynamicTrayBehaviour` in the hierarchy of Figure 2 are

[3] Since we only cover structurally object–oriented models no methods exist. See Section 3.3 for the problems introduced by methods.

therefore used to describe the various issues the modeler must keep in mind when designing a mathematical model for a distillation column tray. Each issue requires a decision on the behaviour of the model to be made more or less independently of the other ones. `TrayVaporFlow` and `TrayLiquidFlow` determine in which way the flow of liquid and vapor into and out of the tray is calculated. `TrayGeometry` contains different geometrical effects influencing the model. `TrayLoading` describes needed additions if the amount of liquid or vapor is much more or much less than the tray was made for. The choices in these areas express different tried choices for each issue in modeling a tray. `TrayTransversalFlow`, for example, contains additional specifications for some balance-equations and variables in case the transversal flow of the liquid in the tray should not be ignored.

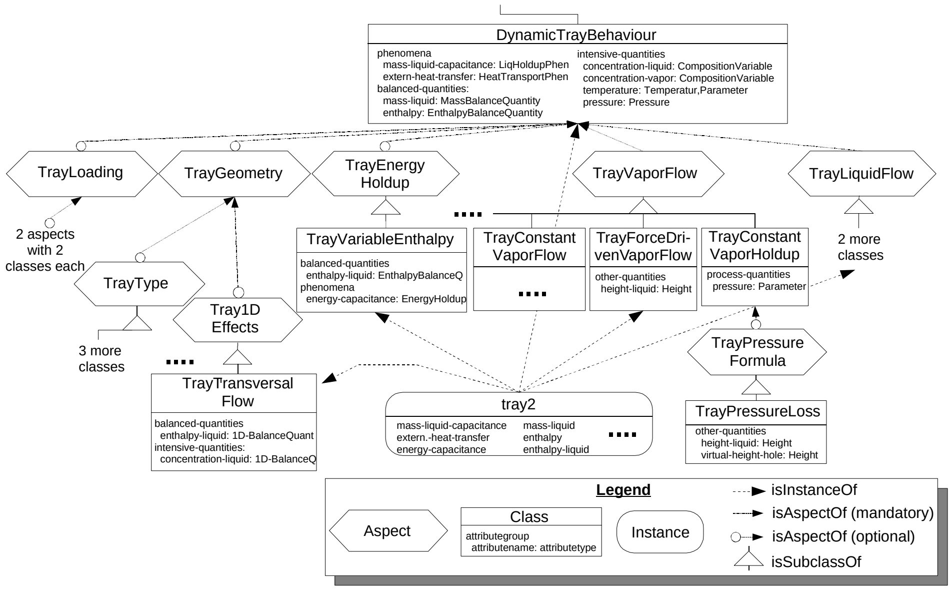

Fig. 2. Conceptual models of a distillation column tray (simplified), using an extended OMT notation

By using and combining these choices the user may select the combination of occurring effects and modeling assumptions fitting his needs. The instantiation relationships of instance `tray2` specify e.g., that a rather detailed model should be generated for tray `Tray2` of Figure 1. This ability to recombine empowers a modeller to reuse class parts and attributes (in our example to provide clues to the equation generator), even if these parts were originally designed in another combination. Also the user can rely on further expert knowledge captured in the hierarchy to avoid mistakes early; for example, pressure loss may only be modelled if vapor holdup is considered also.

Using the class hierarchy as a knowledge repository like the one described above is therefore quite advantageous for the user. It speeds up model development by allowing the use of tested solutions and increases the quality by ensuring that known aspects are not forgotten. On the other hand, such a hierarchy is extremely taxing for 'standard' object models featuring only single instantiation and multiple inheritance for the following reasons:

– Without multiple instantiation all possible combinations of classes have to be materialised as so called intersection classes (cf. e.g. [Ode92]). Mapping Figure 2 into such an hierarchy requires an intersection class for each combination of choices including those combinations without a selection in an optional area. Therefore at least 1152 classes[4] would be needed instead of the 26 classes of Figure 2.
– Furthermore, due to the changing and singleton nature of the application domain, the existing classes will never completely cover all possible models, so new class variations have to be created. But with multiple inheritance some of these changes require either the modification of existing classes or the creation of quite a number of new intersection classes (cf. the "Coloured–Shape" problem posed by [SLU89]).

[4] This number is calculated by multiplying the number of choices in each area leading to (from left to right): $(3 * 3) * (4 * 2) * 2 * (3 + 1) * 2 = 1152$.

– Using just one concept, namely classes, it is difficult to structure large hierarchies like the one described above into issues and alternative choices for these issues.

To summarise, we have seen that representing some of the variation options in a database schema increases its expressivity and understandability. The occurrence of such variety can be recognised in the schema by the existence of many similar classes and especially by diamond-shaped multiple inheritance structures, unless this variety is hidden in the instance level and therefore the knowledge contained in it is not usable at all. There exist two problems in databases with a rich schema originating from representing variation variety: how to minimise the number of classes needed, and how to enrich the structure of the class hierarchy in an intuitive way. The next section describes our solution to these problems for structurally object–oriented data models in more detail. In contrast to other approaches, which are described in Section 4, it is still capable of modeling most of the dependencies existing between different choices, as shown in Section 3.2.

3 Dividing Classes into Aspects

Aspects simultaneously decrease the number of classes and increase the structure of the hierarchy. *Aspects*, represented by hexagons in Figure 2, divide a class into separately instantiatable partitions, similar to the partitions of "multiple classification" proposed by [FS97]. When instantiating an object from a class, one subclass in each aspect is selected by the user to further specify the type of the object. The aspects themselves are just abstract classes and may define attributes and methods if those are valid for all possible subclasses.

If classes are split into aspects, the information distributed among the parts must be joined somehow to gain a schema for each instance. To avoid combinatorial growth of the number of classes, one could delay this join until an instance is created. Each object is then an instance of multiple classes instead of only one (see instance `tray2` in Figure 2). Alas, multiple instantiation has some flaws of its own, including unlimited instantiatability and horizontal name collisions:

Unlimited instantiatability may lead to objects being instances of two semantically incompatible classes like `Variable` and `Equation`, or `Person` and `Vehicle`. In the case of single instantiation this problem is avoided since each object may only instantiate one class, which is (hopefully) consistent. With multiple instantiation one can restrict instantiatability in two ways[5]: Either one specifies permitted transitions between already instantiated classes and those classes which an object may instantiate additionally or alternatively to the instantiated classes (*transitional restrictions*). Or one can restrict the targets of additional instantiation relationships to classes in the structural vicinity of already instantiated classes (*structural restrictions*). The mechanism we propose uses such structural restrictions based on the newly defined "isAspectOf" relationship. Only classes connected by a path of isSubclassOf and isAspectOf relationships to an already instantiated class may be instantiated additionally and the subtree of each aspect may only contain one directly instantiated class.

Horizontal name collisions arise due to the fact that different classes instantiated by the same object may contain attributes or methods of the same name but different type. They also occur with multiple inheritance, where they were described by [Knu88]. In contrast to vertical name collisions — i.e. name collisions between a subclass and a superclass, which are allowed in some object models if the attribute type is specialized — horizontal name collisions are mostly either forbidden or allowed only if a common superattribute exists and and the types of the conflicting attributes have a common subtype. In Subsection 3.2.1 we will present a way of solving name collisions by using the capabilities of multiple instantiation.

3.1 Informal Definition of Aspects

"Aspects" can be defined using the well known OO concepts of classes, objects, instantiation, inheritance and subclassing by introducing an additional relationship "isAspectOf". We give an informal definition here, a formal one can be found in [BM98], using an extension of Jeusfeld's O-Telos semantics [JGJ$^+$95].

Aspect: An aspect is an abstract class with at least one outgoing "isAspectOf" relationship. A class with an incoming "isAspectOf" relationship is called "aspect holder" (see Figure 3a).

[5] Both mechanisms may be combined with logical constraints, which are ignored here, since they add a further layer of complexity.

isAspectOf: isAspectOf is a transitive and asymmetric relationship. Acyclicy follows from these requirements.

Instantiation: If an object instantiates a class with mandatory aspects, it must also instantiate each of these aspects[6]. If an object (indirectly) instantiates an aspect, it must also instantiate at least one of it's holders. The resulting implications are shown in Figure 3b.

Restriction of multiple instantiation: If an object instantiates two different classes, the closures (see Figure 3b) of these two instantiation relationships over the subclassing relationship must not contain common classes.[7]

isAspectOf and subclassing: There must not exist both a subclass relationship and an isAspectOf relationship between the same pair of classes.

Attributes of an aspect or one of its subclasses, which also occur in the aspect holder, must follow the same restrictions (i.e. specialization or subtyping) as if a subclass relationship existed between the aspect and the aspect holder. (This last restriction is not obligatory for aspects to work, but ensures more intuitive hierarchies and an equivalence between single aspect children and a subclass as shown in Figure 3c. Without it a mechanism similiar to the one described in Section 3.2.1 has to be used for resolving the vertical name conflicts.)

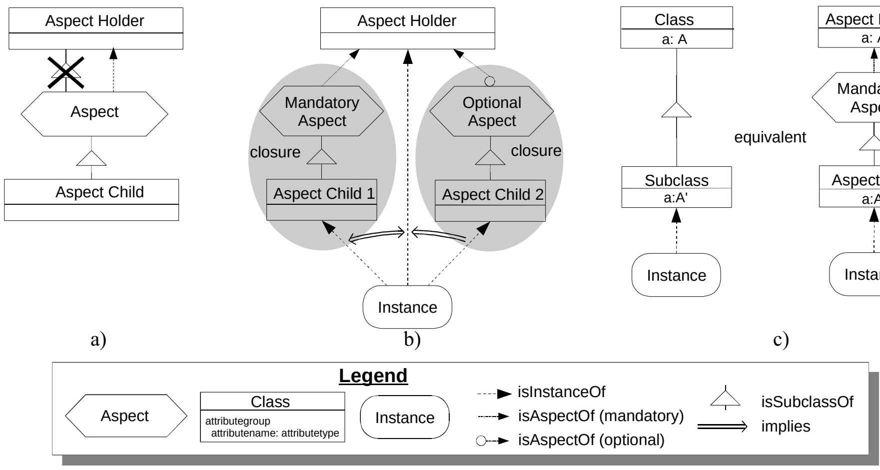

Fig. 3. Illustrations of definition

3.2 Modeling Interdependencies

When describing the example of Figure 2, we ignored dependencies between aspect children to simplify matters. But normally such dependencies are likely to occur, if one tears apart classes according to different aspects. For example, the classes `TrayTransversalFlow` and `TrayVariableEnthalpy` both contain an attribute named `enthalpy-liquid`. In the more commonly know "Person with roles"–scenario (cf. [WdJS95, GSR96, WCL97]) such dependencies also exist: The role "married" is mutual exclusive with the role "catholic priest", the role "in jail" restricts the number of available employment roles quite a bit and, the role "lecturer" requires the role "Ph.D.". Also the attribute "works at" of a "laboratory assistant" is of the type "laboratory" whereas that of a "chemist" is of the type "chemical facility". Thus, a person being a "chemist" and a "laboratory assistants" works at a place being both a "laboratory" and a "chemical facility", i.e. a chemical laboratory. Summing up, we identified the following kinds of possible dependencies:

1. Interdependencies between attributes in different class parts
   (a) value–related dependencies between attributes of the same name

[6] Since aspects are abstract classes and therefore may not be instantiated directly, the object must instantiate a subclass of each mandatory aspect.

[7] This restriction is only sufficient if the hierarchy is a connected graph, i.e. if a root class exists. If no common root class exists, we must furthermore require that all closures except one contain an aspect. The one closure without an aspect in it is the closure over the aspect holder.

   (b) type–related dependencies between attributes of the same name
   (c) dependencies between attributes of different names
2. Interdependencies of the instantiatability of class parts
   (a) mutual exclusion of parts (e.g. the "priest vs. married" example)
   (b) implication of parts (e.g. the "lecturer ⇒ Ph.D." example)
   (c) restriction of applicable choices (e.g. the "in jail, employment" example)
3. Interdependencies between attribute values and instantiatability (e.g. the choice "solid" of a substance is only valid if the substances is below its melting–point.)

Value–related interdependent attributes in different classes influence each other's value in a common instance, whereas type–related ones influence each other's type. Interdependencies between attributes of the same name are quite similar to the problem of horizontal name collisions already mentioned on page 4. We propose a way of handling them in the next subsection. Dependencies between attributes of different names are not handled by our approach, since we disregard constraints, which could solve most cases. The same holds for interdependencies between attribute values and instantiatability of aspect children.

Instantiatability dependencies restrict the combinations of class parts according to some domain knowledge. Reasons for such a restriction can be of different natures: More structural ones like the different alternative choices for the same issue (which we already handle by grouping those choices into the same aspect), more semantic ones like those mentioned at the beginning of this section, or even quite domain dependent ones like e.g. the unsolvability of the simulation equations if inappropriate modeling choices are combined. We will tackle instantiatability dependencies in Subsection 3.2.2.

3.2.1 Attribute Interdependencies: Name Conflict Resolution

The definitions in Section 3.1 allow an object to instantiate two classes containing the same attribute, i.e. a horizontal name collision may occur. What does this mean for the type and value of the attribute? As previously said, current approaches normally prohibit this situation or solve it by assigning an individual scope to each of the classes. This may be caused by the assumption that equally–named attributes defined in different classes actually designate different things. Since aspects are based on the idea that a class is split into parts, we assume the opposite, i.e. that equally–named attributes denote the same thing.

In Figure 4a) attribute `a` is an example of a conflicting attribute. To determine the most natural type for `a` in this situation, let's assume we were using multiple inheritance instead of multiple instantiation. `a`'s type in the intersection class then consisted of the types `T` *and* `S`, i.e. `a` would be in the intersection of both types (Figure 4b). Object systems without multiple instantiation normally require a most specific type for each object and each attribute and do not allow multiple typing. The definition `a:S,T` would be invalid in such a system. But since we are using multiple instantiation anyway, this type conflict can be solved easily

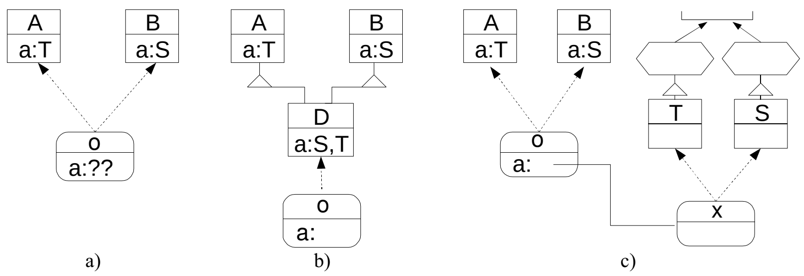

Fig. 4. Name collision, the abstract way: a) with multiple instantiation; b) represented with an intersection class; c) solved with multiple instantiation of the attribute target object

by allowing multiple types for an attribute. A multi–typed attribute requires that the instance it refers to instantiates two (or more) classes which jointly fulfill the type requirements[8]. In the above example the instance `x` referred to by the attribute `a` of `o` has to be an instance of both class `S` and class `T` to fulfill the type required by the multiple instantiation relationships of `o` (cf. Figure 4c). In this way aspects allow to

[8] It is assumed, that all types are classes. If primitive types like "int" and "char" exist, special precautions have to be taken.

refine the type of an attribute without access to the class and its subclasses if one is allowed to define a new aspect (*type restriction by name collisions*).

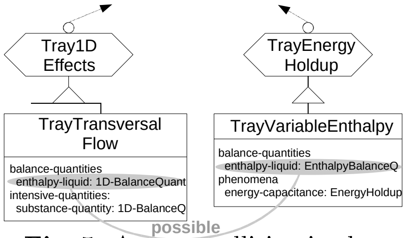

Fig. 5. A name collision in the example

The solution for the type conflicts of equally–named attributes given above also implies a solution for the value dependencies between them: All equally–named attributes in an instance must have the same value. This is quite in line with viewing the attributes as denoting semantically the same thing. Minor problems can only occur when the attributes describe this "thing" in different ways, e.g. in different units. In our application, for example, this is handled by representing the unit as an explicit attribute of a value instead as implicit annotations of the attribute referencing the value.

3.2.2 Instantiation Dependencies

The instantiatability of an aspect child may depend on other aspect children which were already instantiating the object in question. The dependencies to other aspect children of the *same* aspect has already been described as mutual exclusion, which is enforced automatically by "Restriction of multiple instantiation"–axiom of aspects. Thus, we will now consider how the dependencies between children of *different* aspects as identified at the beginning of Section 3.2 can be modeled.

Mutual exclusion

Mutual exclusion of aspect children can not only happen within an aspect. Furthermore children of different aspects may exclude each other, too. If, for example, we modeled the liquid mass holdup separately, as shown in Figure 6, `TrayConstantVaporHoldup` and `TrayVariableVaporFlow` would end up in

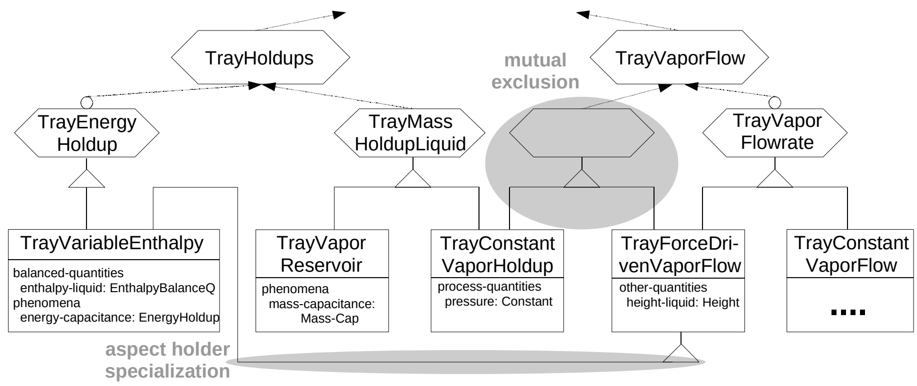

Fig. 6. Aspect holder specialization and mutual exclusion

different aspects. Nevertheless they may not be instantiated together, since a variable flow (as implied by `TrayVariableVaporFlow`) leads to a variable holdup. This can be modeled by introducing a further aspect (the grey–colored one), containing both classes as children. Since there exists no domain equivalent for this aspect, it remains unnamed.

Aspect implication — aspect holder specialization

Instead of being incompatible with another aspect child as described above, an aspect child may require another class to be instantiated. The aspect child therefore implies another one, as in the "lecturer–Ph.D."–example. If this implication holds in both ways, i.e. both aspect children can only be instantiated together, the model is probably wrong and both classes should be merged into one. Otherwise the implication can be modeled by making the implying aspect child a subclass of the implied one. In this way the implied child is automatically instantiated if the implying one is. We call this *aspect holder specialization*, since it makes a statement on additional features which the aspect holder must have for an aspect to be valid. Despite this name the implied class need not always be a subclass of the aspect holder, but may also be another aspect child of it.

This way of modeling is in agreement with the definition given above, since it does not forbid a class to be a member of more than one aspect. In this case a choice in both aspects is made by instantiating the implying class.

Figure 6 shows an example for an aspect holder specialization. The class `TrayVariableEnthalpy` should only be chosen if we also model a variable flow between the trays. Otherwise an index–problem might occur in the mathematical equations generated from the object. By making `TrayVariableEnthalpy` a subclass of `TrayForceDrivenVaporFlow` we ensure, that this problem can not occur.

Restriction of applicable choices — Aspect specialization
The selection of certain choices (i.e. children) in one aspect may restrict the applicable choices in other aspects. In the beginning of this section we gave the "in jail–employment choices"–example for this situation, we will now look at it in the context of our overall example. In Figure 7 if we take account of the effects of weeping[9] by instantiating the class `Weeping`, we want to restrict the choices for the aspect `TrayType` to those types on which weeping can occur. None of the classes implies the other, since we may model a sieve tray without regarding weeping and we may model weeping without a decision to be made which type of tray is actually used.

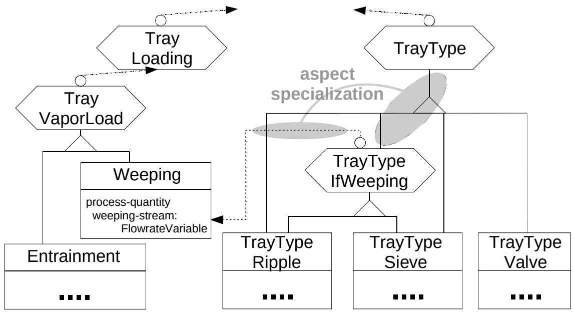

Fig. 7. Aspect specialization

This kind of relationship can be modeled by a specialized aspect, i.e. an aspect which is also a subclass (or more correct a sub"aspect") of another aspect, attached to the restricting aspect child. Instances of `Weeping` may then only choose among the children of its corresponding aspect `TrayTypeIfWeeping` instead of among those of `TrayType`. Alas, the definition of aspects has to be slightly extended to handle aspect specialization.

Aspect specialization: If an aspect A (`TrayTypeIfWeeping` in the example) is a subclass of another aspect (called "superaspect" (`TrayType`)), then any instance of its aspect holder (`Weeping`) may only instantiate the superaspect of A if it also instantiates A itself.

Instantiation (revision of 2nd sentence): If an object instantiates an aspect it must either instantiate one of the aspect holders of the aspect or, if that exists, a superaspect of it.

Please note that no additions to the definition of aspects are needed to model the other kinds of dependencies.

3.2.3 Problems of Dependency Modeling

Whereas the instantiation dependencies, which we presented, increase the modeling power modeling them as shown also somewhat obfuscates the resulting hierarchy. More important, however, is the question of efficiency. In the presence of many restrictions but few choices the representation of different combinations using aspects is less efficient than the one using intersection classes. Figure 6, for example, contains 11 classes and aspects but only models 4 choices[10]. So, while aspects allow many possibilities to model interdependencies, the actual usage has to be weighed against the clarity of the resulting hierarchy.

[9] Weeping denotes the raining of liquid through the holes in the tray if the vapor pressure is too low.
[10] In detail: `ConstantVaporFlow` & `ConstantVaporHoldup`; `ConstantVaporFlow` & `VaporReservoir`; `ForceDrivenVaporFlow` & `VaporReservoir`; `ForceDrivenVaporFlow` & `VaporReservoir` & `VariableEnthalpy`

3.3 Aspects and Methods

Since our project aimed at designing a data model and a schema for a central repository of chemical process models, methods have been somewhat disregarded. Aspects and aspect children may define methods like any other class, but in contrast to attributes no signature conflicts are allowed between methods in different aspects. It is therefore not possible to automatically combine methods of different aspect childs.

Combining methods on the fly from more or less independent parts in a way similar to the mechanism aspects provide for classes and attributes have been recently explored by other authors but seems to be incompatible with aspects: [Mez97] describes a mechanism for decorator–like, i.e. incremental, composition of behaviour. It uses an invisible proxy class for each object, called metaCombiner, to dispatch method calls according to the scope of the class they are defined in. There are two incompatibilities to our approach: First, metaCombiners are introduced to keep up the encapsulation of classes, whereas aspects explicitly break encapsulation to merge equally–named attributes. Second, aspects do not define any incremental order when combined, whereas metaCombiners work best with incremental additions of behaviour. [Pre97] proposes the composition of objects and methods from so called "features". A pair of features is combined using a "lifter" which adapts the methods of one feature to the context of the other. An arbitrarily fixed combination order is defined on the features so only one lifter has to be defined for each pair. Due to the necessity of lifters for each possible combination pair this solution becomes somewhat contrary to the aims of aspects which try to reduce the number of classes.

3.4 Applications

Besides developing some example models, our approach has so far mainly been applied to re-organize the knowledge captured in the VeDa class schemas in a more economic manner. These base classes represent the knowledge gained in numerous modeling efforts over the past decade and form the conceptual basis for a process engineering data warehouse under development in the CRC IMPROVE at RWTH Aachen. A small excerpt of this model is shown in Figure 8. It demonstrates how attribute type interaction allows to easily model the consistency constraints concerning the coupling of interfaces between chemical engineering devices and their inter-connections at any level of aggregation.

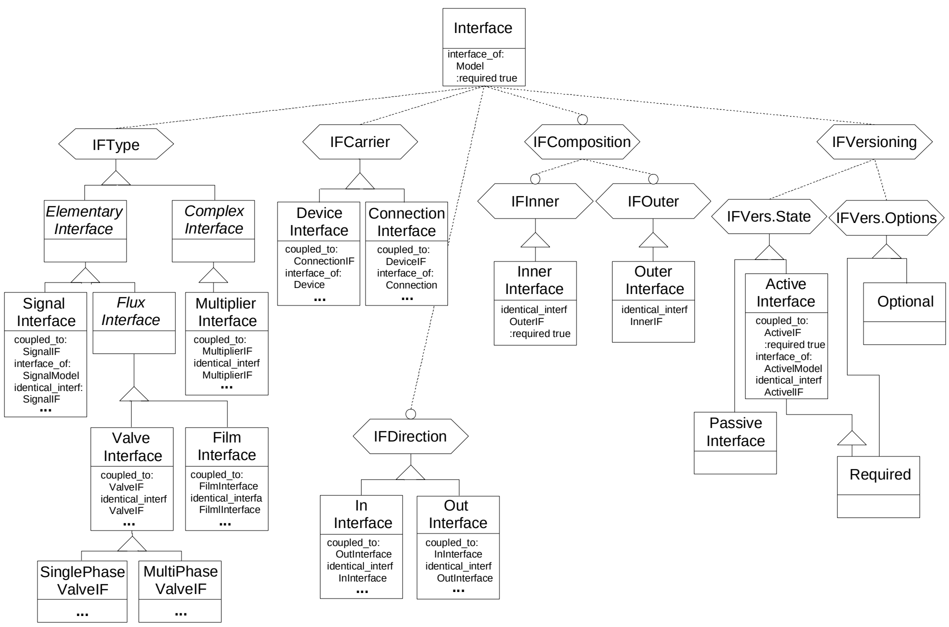

Fig. 8. Modeling device interfaces with aspects

Creating a model with aspects is an easy task if a similar model with a complete set of intersection classes for all possible combinations can be used as a guideline. The `IFType`, `IFCarrier` and `IFDirection` aspects of Figure 8 have been created in such a way. Less abstract models, like the one shown in Figure 2 are more difficult to create, since they contain more instantiation dependencies and models perfectly valid from a domain point of view give unsightly results when modeled with aspects (like the variant shown in Figure 6).

In terms of modeling methodology, aspects suffer from the same problem as other mechanisms allowing multiple instantiation: There exists no clear rule when to use multiple instantiation and when to use aggregation. Modelling "Education" in the "Person" example as an aspect allows to ask a "Person" object directly about the universities it visited, while modelling education as an attribute of Person eases the modelling of the dependencies between "Job" and "Education". Therefore, the TechMod implementation of our work embed the data model described here in a process-integrated modeling environment, offering guidance and traceability [DLP+96].

3.5 Implementation

Like the initial C++–based TechMod version of the VeDa data model, the aspect extension of the object model has been implemented as an object layer on top of the RDBMS Sybase, which provides industrial-strength persistence services. The RDBMS contains the object–oriented schema data and a relation for each class of the schema to save the instance parts in it. The object layer provides services for the access and manipulation of objects and classes used by the applications. For efficiency reasons, the object layer distinguishes between simple and multi objects, only the later may use multiple instantiation. Multi objects are implemented as a proxy object containing pointers to all object parts and an attribute location cache to speed up attribute access.

4 Related Work

Several methods have been proposed to reduce the number of classes in an object-oriented schema, or to enrich the class hierarchy more knowledge structure. We first discuss the closest approach, classes with roles, in some depth, then briefly summarize other related techniques.

4.1 Aspects versus Roles

Roles represent a mechanism to structurally restrict multiple instantiation similar to aspects. Roles have been described by different authors — [WdJS95, GSR96, AGO95, RS91] to name just a few — in many variations and under different names. These variations differ in whether roles have an own object identity, whether the same role can be instantiated more than once per object, whether the role space is partitioned, which method lookup strategy is used, whether transitional instantiation restrictions can be defined, and whether roles can restrict or hide attributes of the classes. But they all share the solution to the problems of unlimited instantiatability and horizontal name collisions: Roles restrict multiple instantiation to 'sub'roles of already instantiated classes, which is virtually the same strategy aspect use. But none of the role mechanisms distinguishes between the region of choice and its alternative choices, and only some allow mutual exclusion to be modeled.

Roles avoid horizontal name collisions altogether by confining the scope of attributes to the roles in which they are defined and to subclasses/subroles of these roles. This requires that each access to an attribute knows the role (e.g. by using a pointer to it or to an subrole) whose scope it should use. It also prevents the kind of attribute type specialization we presented in Section 3.2.1.

The ability of roles to instantiate the same role more than once (i.e. a person may be project manager of more than one project), which is tightly coupled with the isolation of the roles against each other, seems to be unnecessary and counterproductive for our application.

[WCL97] describe a way to specialize the "role player" (similar to what we call aspect holder) of a role to a subclass of the original role player. Their work initiated our development of "aspect holder specialization", which is quite similar to their "role player specialization". We have additionally shown a way to specialize the aspect child in subclasses of the aspect holder ("aspect specialization") in this paper.

The term "aspect" has already been used by [RS91]. But they define those aspects very similar to roles: Different aspects of an object are accessed by different pointers, they are mutual invisible and may hide parts of the aspect holder. We therefore took the liberty to use "aspect" for our methodology.

Similar to our position, albeit from an OOA point of view, [Kat97] criticizes that the methodology of roles allows to model only "role like" relationships. His solutions, "assimilation" and "property modeling", allow any kind of class decomposition. "Assimilation" is realized by introducing invisible delegation links between the instance of an assimilating class and instances of the assimilated classes. He is, however, not considering partitions of the class space and disregards the problems of attribute lookup with multiple delegation, of object identity and of name collisions.

4.2 Other Work

Besides the structural solution offered by roles, there are also a number of methods which try to limit the class hierarchies in a dynamic manner, without changing the object model itself. Class hierarchy reorganization (e.g. [Run92]) can eliminate redundant classes in a hierarchy, but is not able to fundamentally reduce the number of classes. Mixin–Classes [BC90] reduce the need for code duplication and corresponding classes, but still require intersection classes.

[Odb94], [LD94], and others propose ways to transitionally restrict multiple instantiation. Odberg's transitional restrictions allow to model dependencies between attribute values and instantiatability of classes by attaching constraints to the transitions. Nevertheless transitional restrictions seem to be more appropriate for modeling the life–cycles of objects. In our application, where the user may want to randomly switch between different alternatives, they would either restrict him or would result in a tightly interconnected graph.

[NR92, NRE92] reason that standard object–oriented models are inappropriate for several scientific applications. Their modeling language SHOOD contains meta classes, multiple instantiation, and automatic classification. Name conflicts in multiple instantiation are solved by specifying the origin class of the attribute definition when accessing it, resulting in a behavior similar to roles. The authors acknowledge the problem of representing interdependencies among the class parts using their concept of multiple instantiation. As a solution, they propose the combination of aggregation and multiple instantiation. The exact definition of this approach remains unclear in both papers, though.

Orthogonal partitioning of the subclasses of a class has been proposed by [FS97, Mey97] in the context of OOA/OOD and by [WdJS95] in the context of roles. [WdJS95] use roles to implement the multiple instantiation needed for this and therefore can not model attribute interdependencies. The definition in [FS97] focuses on OOA and therefore does not handle points like name collisions and implementation.

Name conflicts and dependencies are also an important topic in distributed database integration. Whereas aspects might be usable for identifying partions of a central database to distribute them, they are not powerful enough to handle all conflicts possible during database integration (cf. [KS91]).

Our approach to allow an attribute to have multiple classes as its type, which effectively means a conjunctions of classes, has been driven much further by [MZ86]. They allow the definition of new classes by arbitrary boolean expressions of other classes, even allowing the NOT operator. The disadvantage of this is of course that even determining inheritance becomes NP–complete.

5 Summary

We have defined the concept of "aspects", a concept that combines multiple instantiation, partitioning of subclasses and a special name collision resolution strategy in a way preferable to other object models. Aspects help to solve the problem of the combinatorial explosion of the numbers of intersection classes, which hamper the use of 'standard' object models in engineering. The resulting class hierarchies are less confusing due to the split of class definitions into regions of choice and the applicable choices. In contrast to other solutions to these problems, aspects still allow the modelling of many interdependencies existing in the intersection classes.

Future work on aspects includes developing a systematic design methodology on which the process integration in the TechMod environment can be based. Such a methodology will on the one hand be based on more practical experiences, on the other on principles such as, e.g. the classification checks given by [WdJS95]. Another relevant extension is a management concept for aspects, including questions of ownership and visibility in a multi-user analysis and design process.

Acknowledgment
This work was partially supported by the DFG under Graduiertenkolleg "Informatik und Technik" and in Collaborative Research Centre (SFB 476) IMPROVE. The example of Section two is based on a project about different mathematical models for distillation column trays by G. Deutsch (supervised by R. v.Watzdorf). J. Hackenberg provided many valuable comments and corrections for the hierarchy of Figure 2. W. Marquardt suggested improvements for the example and the overall structure of the paper.

References

[AGO95] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *VLDB Journal*, 4(3):403–444, 1995.

[AWSL92] Shamim Ahmed, Albert Wong, Duvvuru Sriram, and Robert Logcher. Object–oriented database management systems for engineering: A comparison. *Journal of Object–Oriented Programming*, June 1992.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, October 1990. ACM Press.

[BLM96] R. Bogusch, B. Lohmann, and W. Marquardt. Computer–aided process modeling with modkit. In *Proceedings of Chemputers Europe III*, Wiesbaden, October 1996.

[BM97] R. Bogusch and W. Marquardt. A formal representation of process model eqations. *Computers and chemical Engineering*, 21(10):1105–1115, 1997. extended version of presentation at ESCAPE'95.

[BM98] M. Baumeister and W. Marquardt. The chemical engineering data model VeDa; Part 1: VDDL — the language definition. Technical report, RWTH Aachen, Lehrstuhl für Prozeßtechnik, 1998. (in preparation).

[DLP+96] R. Dömges, B. Lohmann, K. Pohl, M. Jarke, and W. Marquardt. PRO–ART/CE — an environment for managing the evolution of chemical process simulation models. In *Proc. of the 10th European Simulation Multiconference*, pages 1012–1017, Budapest, Ungarn, Juni 1996.

[FS97] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, Reading, Mass., 1997.

[GSR96] G. Gottlob, M. Schrefl, and B. Röck. Extending object–oriented systems with roles. *Transactions on Information Systems*, 14(3), July 1996.

[JGJ+95] M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, M. Staudt, and S. Eherer. ConceptBase - a deductive object base for meta data management. *Journal of Intelligent Information Systems, Special Issue on Deductive and Object-Oriented Databases*, 4(2):167–192, March 1995.

[Kat97] Ravi Kathuria. Improved modeling and design using assimilation and property modeling. *Journal on Object–Oriented Programming*, pages 15–24, January 1997.

[Knu88] Jørgen Lindskov Knudsen. Name collision in multiple classification hierarchies. In S. Gjessing and K. Nygaard, editors, *ECOOP '88, European Conference on Object-Oriented Programming, Oslo, Norway*, volume 322 of *Lecture Notes in Computer Science*, pages 93–109, New York, N.Y., August 1988. Springer.

[KS91] Won Kim and Jungyun Seo. Classifying schematic and data heterogenity in multidatabase systems. *IEEE Computer*, 24(12):12–18, December 1991.

[LD94] Qing Li and Guohzu Dong. A framework for object migration in object–oriented databases. *Data & Knowledge Engineering*, 13:221–242, 1994.

[Mar96] W. Marquardt. Trends in computer–aided process modeling. *Computers and chemical Engineering*, 20(6/7):591–609, 1996.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, N.Y., second edition, 1997.

[Mez97] Mira Mezini. Dynamic object evolution without name collisions. In Mehmet Aksit and Satoshi Matsuoka, editors, *European Conference on Object–Oriented Programming (ECOOP)*, pages 190–219. LNCS 1241, Springer, Juni 1997.

[MZ86] D. McAllester and R. Zabih. Boolean classes. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 417–423, New York, NY, November 1986. ACM Press.

[NR92] G.T. Nguyen and D. Rieu. Multiple object representations. In *Proc. 20th ACM Computer Science Conference*, pages 197–204, Kansas City (Mo), March 1992. ACM Press.

[NRE92] G. T. Nguyen, D. Rieu, and J. Escamilla. An Object Model for Engineering Design. In O. Lehrmann Madsen, editor, *Proceedings of the ECOOP '92 European Conference on Object-oriented Programming*, LNCS 615, pages 233–251, Utrecht, The Netherlands, July 1992. Springer.

[Odb94] Erik Odberg. Category classes: Flexible classification and evolution in object–oriented databases. In Gerhard Wijers, Sjaak Brinkkemper, and Tony Wasserman, editors, *Proceedings of the 6th Conference on Advanced Information Systems Engineering (CAISE)*, LNCS 811. Springer, 1994.

[Ode92] James J. Odell. Dynamic and multiple classification. *Journal on Object–Oriented Programming*, 4(8):45–48, January 1992.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443, Jyväskylä, Finland, 9–13 June 1997. Springer.

[RS91] J. Richardson and P. Schwarz. Aspects: Extending objects to support multiple, independent roles. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 298–307, Denver, Colorado, 1991.

[Run92] Elke A. Rundensteiner. A class integration algorithm and its application for supporting consistent object views. Technical Report ICS-TR-92-50, University of California, Irvine, Department of Information and Computer Science, May 1992.

[SLL92] Tor G. Syvertsen, Frank Lillehagen, and Morten Løvstad. A generic object model for engineering design. In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems*, pages 157–166. Prentice Hall, Hempstead, UK, March–April 1992.

[SLU89] L.A. Stein, H. Lieberman, and D. Ungar. A shared view of sharing: The treaty of Orlando. In Won Kim and F.H. Lochovsky, editors, *Object–Oriented Concepts, Databases and Applications*, pages 31–48. Addison–Wesley, Reading, MA, 1989.

[WCL97] R.K. Wong, H.L. Chau, and F.H. Lochovsky. Dynamic knowledge representation in DOOR. In *Knowledge and Data Engineering Workshop (KDEX'97)*, pages 89–96. IEEE, November 1997.

[WdJS95] R.J. Wieringa, W. de Jonge, and P.A. Spruit. Using dynamic classes and role classes to model object migration. *Theory and Practice of Object Systems*, 1(1):61–83, 1995.